The University of Queensland

Master of Engineering Thesis

Use of Reinforcement Learning To Control The Straight Movement Of An Underactuated Robot

Student Name: Jérémy Augot
Course Code: ENGG7282
Supervisor: Dr. Surya P. N. Singh
Submission Date: May 29, 2019

A thesis submitted in partial fulfilment of the requirements of the
Master of Engineering Degree in Engineering Science (Management)

UQ Engineering

Faculty of Engineering, Architecture and Information Technology

Abstract

This thesis report explains how to control the movement of an underactuated robot using Reinforcement Learning (RL). It presents the theory of RL, the computer model of the robot, the architecture of the code and the two approaches followed to solve the problem.

The simulations are done using MuJoCo in a python environment. They simulate the behaviour of the robot in two different state spaces: the first experiments take place in a discrete state space, where the learning is done using Q-Learning and SARSA(λ) algorithms. The second approach is to treat the problem in a continuous state space and implement REINFORCE.

With such algorithms, it is demonstrated that the initial chaotic behaviour of the robot can be controlled to move accurately in a straight direction. It manages to move slightly faster in the continuous state space, but with more variance. At the end of this report, some suggestions are proposed to go further and improve the learning process.

Contents

1 Context

A lot of research has been carried out on Artificial Intelligence (AI) for the last few decades. The increase in computer power enables the development of more and more complex algorithms, some of them being able to outperform humans. As a matter of fact, a few stories of machines defeating humans made the headlines in the newspapers. Amongst them, the machine called *Deep Blue* defeated for the first time in 1997 the World Chess Champion Garry Kasparov [1] and Google's program *AlphaGo* defeated the world number one Ke Jie at the game of Go in 2017 [2].

Both of these machines rely on *Reinforcement Learning* (RL), a type of *Machine Learning* that consists of making a robot learn by itself by interacting with an environment. To do so, the robot - named the *agent* - experiments different actions by trial and error. For each action taken, it receives from the environment a reward or a punishment. The principle of RL is for the agent to take the optimal action for each state that maximizes the reward [3]. In this way, RL enables machines to find optimal behaviours by themselves, becoming more and more intelligent by training and therefore being capable of outperforming humans.

Such a potential to converge to optimal solutions is promising for robotics which usually involves complex behaviours, difficult to control. **Therefore, it is legitimate to wonder if RL can be applied to solve problems driven by complex physical laws that are hard to predict.** An example of a complex behaviour can be found in a simple toy: a Katita - illustrated in figure 1.

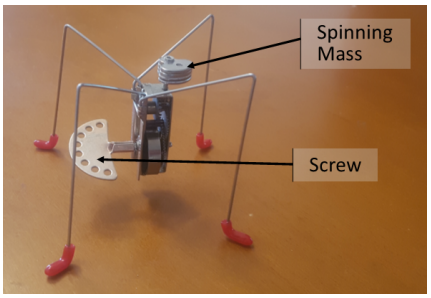


Figure 1: Illustration of the robot

A Katita is a small toy usually used by kids. To make the device work, the user has to twist the screw and release it. By doing so, a spring makes the screw come back to its initial position. This generates a rotation which is transmitted to the mass through gears. The rotation of the mass then makes the whole Katita jump in every direction, by pressing on the legs while spinning.

This robot is underactuated because it has less degrees of freedom than actuators. This results in a very complex behaviour, difficult to predict. However, it is not stochastic and therefore is theoretically controllable by the rotation of the mass.

Thus, can Reinforcement Learning be used to compensate the highly variable motion of an underactuated robot to control it and make it go straight?

---

2 Scope

This paper aims at answering the previous question by proposing a new benchmark of a 3D underactuated robot, along with a set of techniques to control it. More precisely, the goal is to implement Reinforcement Learning to control the movement of the mass so that the robot manages to move along a straight line. However, the work is not directly done on the physical robot, but on a computer model of it.

This paper first explains the Reinforcement Learning background necessary to understand the algorithms used in the following parts. Then, it presents how the robot model was made and how the simulations were carried out. Two approaches were followed to answer the problem: this report compares the performances of the robot acting in a discrete and continuous state space.

At the end, suggestions to improve the learning are mentioned. However, they are not implemented in this thesis and are left for further work.

3 Background

The theory explained in this chapter is mainly extracted from the book *Reinforcement learning: An introduction* written by Sutton and Barto in 2017 and the *Reinforcement Learning Explained* course from Microsoft available on the *edX* platform. It goes through the concepts necessary to understand how RL is used in the next chapters. However, it does not cover everything in depth and a curious reader should go through the references for more advanced information.

3.1 Principle of Reinforcement Learning

As briefly explained in section 1, Reinforcement Learning consists of making a robot learn from its own experience by trial and error. Figure 2 shows in more details the principle of RL.

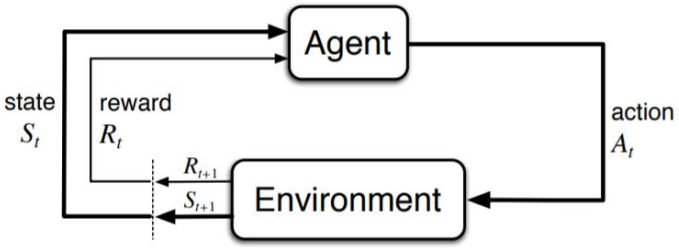


Figure 2: Principle of Reinforcement Learning - [3]

At each time step t, the agent is in a state $S_t \in S$, with S being a set of all the possible states. From this state S_t, it has at its disposal a set of all the possible actions it can take, denoted $A(S_t)$. The action $A_t \in A(S_t)$ chosen by the agent is sent to the environment, which generates the resulting new state S_{t+1} and a reward R_{t+1}. The reward is a real number defined by the code designer that illustrates how good the agent has performed by taking this action. The way to choose an action over another is defined by a *Policy*, usually denoted π. A policy maps each state S_t to an action A_t to take when in that state [3].

Moreover, the agent can either perform an *episodic* or *continuous* task. An *episodic* task lasts a finite amount of time T - i.e. it stops when $t = T$ - whereas a *continuous* task never ends [3].

3.2 Challenges in RL

A. Swaminathan identifies in reference [5] four main challenges in Reinforcement Learning:

- **Exploration:** The first difficulty of RL is to make the agent learn from its previous experiences. Indeed, to do so, it has to take a certain amount of actions to understand which one are leading to a great reward and which one are not. However, how many actions to take before considering that the agent knows what the best behaviour is? What if there is an action still unexplored that could lead to a greater reward? **This is called the exploration/exploitation dilemma:** the designer has to find the trade-off between how many times the agent needs to try new actions - *exploration* - and how many times it has to choose the best action amongst the ones it already tested - *exploitation*.
- **Temporal Credit Assignment:** The agent usually takes several actions before achieving the desired goal. However, how does it know which of these actions taken was salient for the eventual observed outcome?
- **Representation:** Usually, different set of states and actions can be used to model the interaction between the agent and the environment. The amount of states/actions and how well they represent the problem have a huge influence on the learning efficiency. Therefore, how to best define the set of states/actions so that the agent can efficiently learn?
- **Generalization:** The goal of RL is for the agent to learn what the best behaviour is without trying all the actions and states. This leads to the question: is the agent capable of behaving well in unseen states?

These challenges form the core of the RL theory. They are studied more in depth in the following subsections.

3.3 Markov Decision Process

A *Markov Decision Process* (MDP) is a RL problem that follows the *Markov property*, which is when **the state representation is such that the decision of an action to take can be made only by looking at the current state, i.e. without needing the full history of the previous robot behaviour** [6]. If a RL problem happens to be a MDP, then the *Value Functions* and the *Bellman equations* can be defined.

3.3.1 Value Functions

Value functions are one of the most important concepts of RL, because they are the key of the learning process. They rely on the notion of *return*.

The return, written G_t, is the long-term accumulation of rewards starting from time t. It is equal to [6]:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1}$$

With: $\begin{cases} \gamma \in [0,1] : \text{discount factor} \\ T = \infty \text{ for continuing task} \\ \gamma = 1 \text{ for episodic task} \end{cases}$

Knowing that, it is possible to define two types of value functions [6]:

- **State-vale function** that gives the return the agent can expect from being in a given state s and following the policy π. Mathematically, it is defined by:

$$v_\pi(s) = E(G_t|S_t = s) \tag{2}$$

- **Action-value function** that gives the return the agent can expect from being in a given state s and taking the action a to follow the policy π:

$$q_\pi(s, a) = E(G_t|S_t = s, A_t = a) \tag{3}$$

Value functions are important in RL because they provide a criterion to choose an action over another. Indeed, if the expected return from being in a state s and taking the action a_1 is higher than being in that same state but taking the action a_2, i.e. $q_\pi(s, a_1) > q_\pi(s, a_2)$, the agent has more interest in taking the action a_1.

3.3.2 Bellman equation

The definitions of the value functions in equations 2 and 3 are easy to understand, but not convenient to use. The Bellman equation 4 writes the state-value function more explicitly, considering the following notations:

- $\pi(a|s)$: the probability of taking an action a from the state s. It is defined by the policy π.
- $p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$: the probability of ending in a state s' and getting the reward r, by taking the action a from the state s.
- $\gamma \in [0, 1]$: the discount factor.

References [3] and [6] prove that the state-value function can be written as follow:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{4}$$

The Bellman equation is very convenient because it provides a recursive relationship between the current state s and the next one s', which is the key of *Dynamic Programming* (see subsection 3.4).

Figure 3 illustrates what the Bellman equation does.

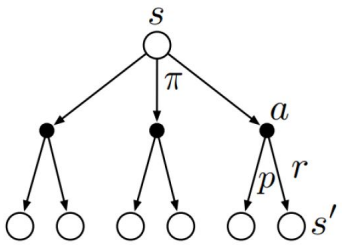


Figure 3: Illustration of the Bellman equation - [3]

From the state s, it explores every possible action a that can be taken by following the policy π. It evaluates the expected return of this action a by exploring all the possible states s' that can be visited by taking this action.

The efficiency of a policy π can be assessed by its value functions v_π and q_π. Therefore, it is possible to compare different policies and define the optimal one.

3.3.3 Optimal Policy

The optimal policy is the one leading to the greatest expected return, and is usually written π_*[6]. In other words, it is the one associated to the optimal value functions [3] [6]:

$$v_*(s) = \max_{\pi}(v_\pi(s)) \tag{5}$$

$$q_*(s,a) = \max_{\pi}(q_\pi(s,a)) \tag{6}$$

3.4 Dynamic Programming

Dynamic Programming (DP) is a set of techniques for solving optimization problems by dividing them into several overlapping subproblems [7]. It relies on the use of value functions to first evaluate a policy and then improve it.

3.4.1 Policy Evaluation

To evaluate a policy π, the idea is to compute the expected return of being in each state, i.e. $\forall s \in S$, compute $v_\pi(s)$ [7]. In practice, this can be done by iterations, by denoting:

$\begin{cases} k\text{: the incremental integer} \\ V_k\text{: the array holding the estimated value of each state at the } k\text{-th iteration} \end{cases}$

The state-value function can be written [3] [7]:

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)(r + \gamma V_k(s')) \tag{7}$$

Therefore, the state-value function can be evaluated by applying algorithm 1.

Algorithm 1: Iterative Policy Evaluation - [3] [7]

Input: policy π to be evaluated
Initialize an array $\forall s \in S\ V(s) = 0$
repeat
  $\Delta = 0$
  for *each* $s \in S$ **do**
    $v = V(s)$
    $V(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)(r + \gamma V(s'))$
    $\Delta = \max(\Delta, |v - V(s)|)$
  end
until $\Delta < \theta$ *(small number defined by the designer)*;
Output: $V \approx v_\pi$

Once the policy has been evaluated, it can be optimized to improve the behaviour of the agent.

3.4.2 Policy Improvement

Policy improvement is based on the following theorem [3] [7]: **for all states s, if following a new policy π' for one step and then going back to the current policy leads to a higher return, then the new policy π' is better than or equal to the**

current policy. Mathematically, it can be written as [7]:

$\forall s \in S$, $q_\pi(s, \pi'(s)) \geq v_\pi(s) \implies v_{\pi'}(s) \geq v_\pi(s)$, with $\pi'(s)$ being the new action taken from the state s, determined by the new policy π'.

Therefore, by applying this theorem, one can be sure that the policy π' that picks for each state the action that maximizes the expected return will be better than or equal to the current policy π. This policy is called the *greedy* policy, and is defined as:

$$\pi'(s) = \arg\max_a (q_\pi(s, a)) \tag{8}$$

It is demonstrated in reference [3] that:

$$\pi'(s) = \arg\max_a (\sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s'))) \tag{9}$$

Knowing this theorem, a policy can be improved by iterations, as presented in the next section.

3.4.3 Policy Iteration

The idea of policy iteration is quite straightforward and is illustrated in figure 4.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

Figure 4: Principle of Policy Iteration - [3]

It consists in alternatively evaluating a policy by using algorithm 1 and then improving it with equation 9, until the optimal policy π_* is found. This is summarized in algorithm 2.

Algorithm 2: Policy Iteration - [3] [7]

1. Initialization
Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily $\forall s \in S$

2. Policy Evaluation
repeat
  $\Delta = 0$
  for *each* $s \in S$ **do**
    $v = V(s)$
    $V(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))$
    $\Delta = \max(\Delta, |v - V(s)|)$
  end
until $\Delta < \theta$ *(small number defined by the designer)*;

3. Policy Improvement
$policyStable = True$
for *each* $s \in S$ **do**
  $oldAction = \pi(s)$
  $\pi(s) = \arg\max_a (\sum_{s',r} p(s', r|s, a)(r + \gamma V(s')))$
  if $oldAction \neq \pi(s)$ **then**
    $policyStable = False$
  end
end
if *policyStable* **then**
  Stop and return $V \approx v_*$ and $\pi \approx \pi_*$
else
  Go to 2
end

This algorithm is really simple to understand: it alternates between policy evaluation and policy improvement. However, the main drawback is that policy improvement cannot be performed before the end of policy evaluation, which can take time to converge. This is why usually the *value iteration* method is preferably used.

3.4.4 Value Iteration

The principle of value iteration is to approximate the state-value function after 1 sweep of policy evaluation, rather than repeating the loop until $\Delta < \theta$. Policy improvement and policy evaluation are combined in one update, as shown in equation 10 [3] [7].

$$V_{k+1}(s) = \max_a (\sum_{s',r} p(s', r|s, a)(r + \gamma V_k(s'))) \tag{10}$$

Instead of looking for the best evaluation of the state-value function to improve the policy, the optimization is directly done on the first approximation of the expected return. This is written in the resulting algorithm 3.

Algorithm 3: Value Iteration - [3] [7]

Initialize $V(s) \in \mathbb{R}$ arbitrarily $\forall s \in S$

repeat
  $\Delta = 0$
  for *each* $s \in S$ **do**
    $v = V(s)$
    $V(s) = \max_a \left(\sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))\right)$
    $\Delta = \max(\Delta, |v - V(s)|)$
  end
until $\Delta < \theta$ *(small number defined by the designer)*;

Output: a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \left(\sum_{s',r} p(s', r|s, a)(r + \gamma V(s'))\right)$

Value iteration is slightly faster than policy iteration because it does not wait for the full evaluation of the policy to improve it. However, both of these algorithms rely on the assumption that the probabilities $p(s', r|s, a)$ are known, which is usually not the case. **Indeed, in robotics, the model of the MDP is very often unknown.** This means that, by taking the action a from the state s, the designer does not know how likely the agent is to end up in a specific state s'. Therefore, these algorithms need to be adapted for model-free MDPs: this can be done through *Monte Carlo Learning* and *Temporal Difference Learning*.

3.5 Monte Carlo Learning

Monte Carlo Learning (MC) is the easiest way to evaluate a policy without knowing the model of the MDP: it uses the concept of empirical mean to approximate the expected return [8]. The idea is presented in algorithm 4.

Algorithm 4: Policy Evaluation by Monte Carlo Learning - [8]

Initialize $V(s) = 0\ \forall s \in S$
Initialize a counter for each state: $\forall s \in S$, $N(s) = 0$
Every time t that the state s is visited in an episode **do**
  Increment the counter $N(s) = N(s) + 1$
  $V(s) = V(s) + \frac{1}{N(s)}(G_t - V(s))$
Output: $V \approx v_\pi$

Although it is really simple to implement, the Monte Carlo method is not efficient because it can only learn from complete episodes. Indeed, the evaluation of the state-value function involves the use of the return G_t, which by definition in equation 1, is the accumulation of rewards starting from time t. Therefore, it needs to go through all the steps of the episode, starting from t, to calculate G_t and estimate v_π. Temporal Difference Learning tackles this issue by approximating the return without having to complete the whole episode.

3.6 Temporal Difference Learning

Temporal Difference Learning (TD) presents a set of methods to solve model-free MDPs. More precisely, it offers an efficient way to evaluate and optimize policies where the state transition and reward functions are unknown.

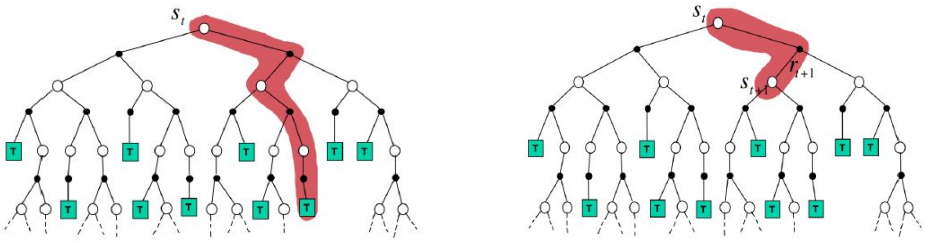
The simplest version of TD learning, called TD(0), relies on the approximation of the return G_t by looking only one step ahead, instead of running the complete episode [3] [8]. In this way, the evaluation of the state-value function presented in algorithm 4 becomes [3] [8]:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) = V(S_t) + \alpha\delta_t \tag{11}$$

With:

- S_t: current state at time step t
- S_{t+1}: state visited at $t+1$ by taking the action from S_t.
- R_{t+1}: reward received by taking the action.
- $R_{t+1} + \gamma V(S_{t+1})$: approximation of the return G_t by looking one step ahead.
- γ: discount factor.
- α: step-size paramter.
- $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$: TD error.

The main advantage of this simple version of TD learning, as illustrated in figure 5, is that it can learn from incomplete episodes by only looking on step ahead. Therefore, it works for continuous tasks that never finish. However, although it is more efficient than MC, it is more sensitive to the initial value [8].



(a) Illustration of MC Backup - [8] (b) Illustration of TD(0) Backup - [8]

Figure 5: Comparison of MC and TD(0) methods

The main drawback of only looking one step ahead to evaluate v_π is the accuracy of the approximation of the return G_t. One could expect that looking 2, 3 or n steps ahead would be more accurate: this is the purpose of the TD(λ) method.

3.6.1 TD(λ) for Policy Evaluation

TD(λ) is a method to increase the accuracy of the approximation of the expected return by looking more than one step ahead. It relies on the principle of n-steps prediction, illustrated in figure 6.

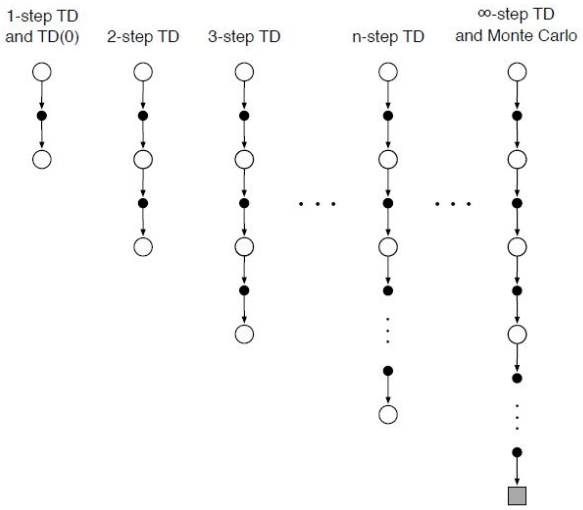


Figure 6: n-Steps Prediction - [3]. 1 step ahead corresponds to TD(0), and looking at an infinite number of steps is the equivalent of MC for continuous tasks

For each case, the return can be approximated by [3] [8]:

- **1 step -TD(0):** $G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1})$
- **2 steps:** $G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})$
- **n steps:** $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$

The n-step TD learning is done through equation 12:

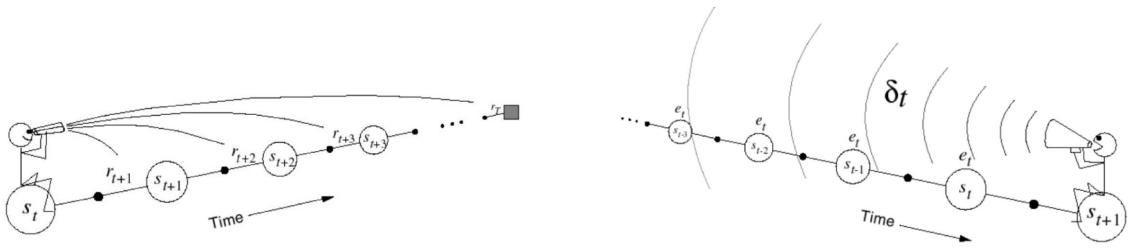
$$V(S_t) = V(S_t) + \alpha(G_t^{(n)} - V(S_t)) \tag{12}$$

To make the approximation of the expected return more robust, **the idea of the TD(λ) method is to combine all the n-step $G_t^{(n)}$ for $n \in \mathbb{N}^*$ by using a parameter** $\lambda \in [0, 1[$. The λ-return G_t^λ is defined by equation 13 [8].

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{13}$$

By averaging all the n-step expected returns, the accuracy of the approximation is greatly increased and becomes way better than the one of the TD(0) method. However, to compute G_t^λ, one has to look into the future, and once again, this can be done only for complete episodes because of n that varies between 1 and infinity. A solution is, instead of looking forward, to look backward.

Backward TD(λ)



(a) Forward TD(λ) - [3]

(b) Backward TD(λ) - [3]

Figure 7: Illustration of forward and backward TD(λ)

As illustrated in figure 7, backward TD(λ) looks at the previous states instead of looking into the future. More precisely, to know which state had a significant influence on the agent behaviour, it uses the notion of *Eligibility Trace*. It is a function for temporal credit assignment that gives credit to [8]:

- The most frequently visited state
- The most recently visited state

Mathematically, this function is defined, for each state $s \in S$, by $E_0(s) = 0$ and :

$$E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s) \tag{14}$$

γ and λ are parameters that make the eligibility trace decrease over the time steps. When a state is visited, $1(S_t = s) = 1$ and the function increases. This is illustrated in figure 8.

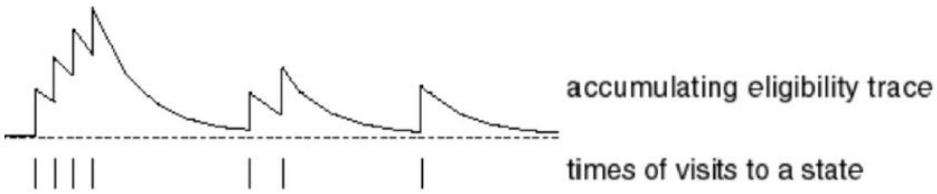


Figure 8: Example of the evolution of the Eligibility Trace for a given state - [8]. One vertical line is drawn each time the state is visited.

Knowing that, it is possible to write the backward TD(λ) algorithm for policy evaluation.

Algorithm 5: Policy Evaluation with Backward TD(λ) - [8]

Initialize $E_0(s) = 0 \ \forall s \in S$
Initialize a counter for each state: $\forall s \in S$, $N(s) = 0$
for *each time step* t **do**
  Update the eligibility trace for every state s:
  $E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s)$
  Compute the TD-error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
  Update the state-value function $V(s)$ for every state s:
  $V(s) = V(s) + \alpha\delta_t E_t(s)$
end
Output: $V \approx v_\pi$

In the end, the backward TD(λ) algorithm happens to be more efficient to evaluate a policy than the forward version [8]. Moreover, the main advantage is that it can be used for incomplete episodes whereas forward TD(λ) cannot. Another alternative solution for continuous tasks is to use the n-step TD learning to approximate the state-value function, even though it is less robust than forward TD(λ).

Now that several methods to evaluate policies of model-free MDPs have been defined, it is interesting to compare how efficient they are regarding policy optimization. There are two ways to improve the policy of a model-free MDP:

- *On-Policy Learning:* it consists of improving a policy π from the experience sampled from this same policy π [8].
- *Off-Policy Learning:* it consists of improving a policy π from the experience sampled from another policy μ [8].

3.7 On-Policy Learning

On-Policy Learning is a "learn on the job" type of learning [8]. A policy improvement algorithm has already been designed in section 3.4 (algorithm 2). However, this algorithm relies on the use of the state-value function. The problem is that the greedy policy defined in equation 9 is calculated with the probabilities $p(s', r|s, a)$. These probabilities are unknown in model-free MDPs, which makes the algorithm unusable. When the model is unknown, one has to improve the policy over the action-state value function $q_\pi(s, a)$ rather than $v_\pi(s)$. Indeed, the best policy is defined as taking the action that leads to the greatest expected return from each state. Mathematically, this can be written as:

$$\pi'(s) = \arg\max_a \left(q_\pi(s, a)\right) \tag{15}$$

Thus, using $q_\pi(s, a)$ instead of $v_\pi(s)$ to optimize a policy allows to avoid the use of the unknown probabilities $p(s', r|s, a)$. As explained in section 3.4.3, the convergence to the optimal policy is done by alternatively evaluating a policy and improving it. The evaluation is done using *TD Control*.

The idea is simple: for every episode, the approximation of the action-value function q_π is done by TD Learning [3] [8]. Then, the optimization is done through ϵ-greedy improvement, which consists of [8]:

- Choosing the optimal action with a probability of $1 - \epsilon$, with $\epsilon \in [0, 1]$.
- Choosing a random action with a probability of ϵ.

The goal of sometimes picking a random action is to prevent the algorithm from being stuck in the first action tried without testing the others [8]. Indeed, if the first action tried seems to be quite efficient, its action-value function will be higher than the one of the other untried actions. Therefore, the next time the algorithm reaches the same state, it will pick this same first action without trying the others.

As there are several types of TD learning algorithms, the policy evaluation can be done by using the following control methods:

- TD(0)

- Forward TD(λ)
- Backward TD(λ)

SARSA
The SARSA algorithm uses the TD(0) approximation of the return to compute the action-value function. The name "SARSA" comes from figure 9.

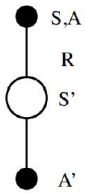


Figure 9: Illustration of the SARSA algorithm - [3].

From the state S, the action A is taken. This leads to a new state S' and the reward R. Then, from S', a new action A' is taken and so on. $Q(S,A)$ is calculated by using the action-state value of the next state/action pair (S',A'), as shown in equation 16.

$$Q(S,A) = Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A)) \tag{16}$$

SARSA(λ)
The idea of the SARSA(λ) algorithm is to use the TD(λ) learning to update the action-state value function $q(s,a)$. This can be done in two ways: either through equation 12 by replacing $V(S_t)$ with $Q(S_t,A_t)$ - this is forward TD(λ) - or by using the eligibility trace defined in equation 14 - this is backward TD(λ). The second method is detailed in algorithm 6.

Algorithm 6: Backward SARSA(λ) Algorithm for On-Policy Control - [3] [8]

```
Initialize Q(s,a), ∀s ∈ S, ∀a ∈ A(s), arbitrarily
for each episode do
    E(s,a) = 0, ∀s ∈ S, ∀a ∈ A(s)
    Initialize S, A
    for each step of episode do
        Take action A, observe R,S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        δ = R + γQ(S',A') − Q(S,A)
        E(S,A) = E(S,A) + 1
        for all s ∈ S, a ∈ A(s) do
            Q(s,a) = Q(s,a) + αδE(s,a)
            E(s,a) = γλE(s,a)
        end
        S = S'; A = A'
    end
end
Output: Q ≈ Q*
```

3.8 Off-Policy Learning

Off-Policy Learning is a "Look over someone else's shoulder" type of learning [8]. In other words, it follows a *behaviour policy* μ to converge to a *target policy* π [8].

One of the most used off-policy learning method is the *Q-Learning Algorithm* [8]. This algorithm is very convenient because it approximates the optimal action-value function q_{π_*} independently of the policy μ that is followed [3], through equation 17 [3] [8].

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a (Q(S_{t+1}, a)) - Q(S_t, A_t)) \tag{17}$$

Therefore, by looking at all the possible actions a that can be taken from the next state S_{t+1}, and returning the maximum expected return $\max_a (Q(S_{t+1}, a))$ of all these action/state pairs, the update $Q(S_t, A_t)$ will converge to the optimal action-value function q_* [3]. The resulting algorithm is as follow.

Algorithm 7: Q-Learning Algorithm - [3] [8]

Initialize $Q(s, a), \forall s \in S, \forall a \in A(s)$, arbitrarily, and $Q(terminal - state, .) = 0$
for *each episode* **do**
  Initialize S
  for *each step of episode* **do**
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R,S'
    $Q(S, A) = Q(S, A) + \alpha(R + \gamma \max_a (Q(S', a)) - Q(S, A))$
    $S = S'; A = A'$
  end
end
Output: $Q \approx Q_*$

Q-Learning and SARSA(λ) are powerful algorithms and converge quickly to the optimal action-value function [3]. **Although they are really well suited for discrete state spaces, they can become inefficient when the problem is represented by continuous states** [9]. This is when another type of learning becomes interesting: *Policy Gradient Learning.*

3.9 Policy Gradient Learning

Unlike TD and Q-learning, Policy Gradient learning does not rely on value functions but approximates directly the policy with a set of parameters θ. Policy optimization is done by finding θ that maximizes the expected sum of rewards at every time step [9]. **The main advantage of this method is that it is very suitable for high dimensional or continuous state and action spaces** [9].

3.9.1 Policy Gradient Derivation (PGD)

Policy Gradient methods rely on the definition of trajectories. A trajectory τ is a sequence of states and actions that are encountered by the agent while interacting with the environment [9]. The associated return is defined as [9]:

$$R(\tau) = \sum_{t=0}^{T} R(s_t, a_t) \tag{18}$$

With $R(s_t, a_t)$ being the reward of taking the action a_t from the state s_t. Following a policy π leads to a set of possible trajectories τ, some of them being more likely to occur than others depending on the policy parameters θ. Therefore, by denoting $P(\tau, \theta)$ the probability of observing the trajectory τ, the expected return of following a given policy π is defined as [9]:

$$J(\theta) = E(\sum_{t=0}^{T} R(s_t, a_t), \pi_\theta) = \sum_{\tau} P(\tau, \theta) R(\tau) \tag{19}$$

The goal of PGD is to find θ that maximizes the expected return by making the trajectories that lead to high rewards more likely to happen. Mathematically, this can be written as [9]:

$$\max_{\theta} (J(\theta)) = \max_{\theta} (\sum_{\tau} P(\tau, \theta) R(\tau)) \tag{20}$$

In practice, the parameters θ are updated by computing the gradient of the expected return $\nabla J(\theta)$ [9]:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{\tau} P(\tau, \theta) R(\tau) \tag{21}$$

After simplification, it is proven in reference [9] that the gradient $\nabla J(\theta)$ can be expressed as a function of actions and states:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T} \nabla_\theta log(\pi_\theta(a_t|s_t)) R_t \tag{22}$$

With:

$\begin{cases} \pi_\theta(a_t|s_t) \text{: Probability of taking the action } a_t \text{ from the state } s_t \text{ at time step } t \\ R_t = \sum_{t'=t}^{T} \gamma^{t'} r_{t'} \text{: The discounted return at time step } t. \\ r_{t'} \text{: Reward observed at time step } t' \end{cases}$

3.9.2 REINFORCE Algorithm

Equation 22 can be used to update the policy parameters θ in order to converge to the optimal policy through the following algorithm.

Algorithm 8: REINFORCE Algorithm [9]

Initialize the policy parameters θ
for *each episode* **do**
  Initialize the first state s_0
  for *each step* $t \in [0, T]$ *of episode* **do**
    Use the current policy π_θ to select the action a_t to take from the state s_t
    Take action a_t, observe r_t
    Store (s_t, a_t, r_t)
  end
  $\nabla_\theta J(\theta) = \sum_{t=0}^{T} \nabla_\theta log(\pi_\theta(a_t|s_t)) R_t$
  $\theta = \theta + \alpha \nabla_\theta J(\theta)$
end
Output: The parameters of the optimal policy $\theta \approx \theta_*$

3.10 Summary

This chapter presents many concepts: the principle of Reinforcement Learning and its challenges, the Markov Decision Process, Dynamic Programming, Monte Carlo Learning, Temporal Difference Learning, On/Off-Policy Learning and Policy Gradient Learning. Not all of these notions are used in the following sections and only algorithms 6, 7 and 8 can efficiently make the agent learn how to move along a straight line. However, although unused, all these concepts are necessary to understand why these algorithms are efficient and are relevant for this thesis.

Indeed, the robot trying to move straight can be represented as a Markov Decision Process with continuous action and state spaces. From there, the problem can be solved through two approaches:

- Discretizing the state space to simplify the representation and apply Q-Learning and SARSA(λ). This approach is less representative of the real problem but can easily solve it. See chapter 8.
- Treating the problem as a continuous problem to apply the REINFORCE algorithm. See chapter 9.

4 Related Work

Reinforcement Learning is a very fast-growing area of Artificial Intelligence and the theory goes far beyond the concepts explained in section 3. It is impossible to present in depth all the algorithms and papers that have been published. Therefore, this section only aims at giving a quick overview of RL research and its applications so that a curious reader knows where to find more information.

Most researches try to develop efficient algorithms that can be applied to complex problems, such as *Games* or *Robotics*. For this purpose, OpenAI created the *Gym* software library which provides a set of benchmark problems for Reinforcement Learning research [10]. It offers several types of environments like *toy control*, *algorithmic*, *Atari games*, *board games* like game of Go and *2D and 3D robots* [10]. More specifically, the set of available robots can be found on the Gym website `https://gym.openai.com/envs/#mujoco`. RL papers often use this library to apply algorithms on concrete problems.

Games
Most games like Go or Atari Games can be represented by discrete action and state spaces. The papers [11], [12] and [13] present several algorithms to efficiently deal with high dimensional spaces. More precisely, [11] introduces the latest version of AlphaGo - *AlphaGo Zero* - which uses a *Deep Neural Network* to approximate the policy with a *Monte Carlo Tree Search* to select actions (see [11] for more details). In addition, reference [12] explains how to apply *Deep Q-Learning* to Atari games and [13] how an agent can learn to play Pacman.

Robotics
Robotics problems are usually represented by continuous state spaces with discrete or continuous actions. The following papers explain how to deal with such complex representations using algorithms like:

- *Q-Learning*, *Actor Critic Policy Gradient* and *SARSA* for problems like Cart-Pole [14], ball collecting tasks [15] or robot navigation between obstacles [16].

- *Deep Deterministic Policy Gradient* presented by DeepMind adapting Deep Q-Learning for complex continuous action spaces [17].
- State-of-the-art policy optimization methods like *Trust Region Policy Optimization* [18] [19], *Proximal Policy Optimization* [18] or *Monotonic Policy Optimization* [4] used to control the motion of 3D robots.
- Supervised Reinforcement Learning [20] [21].

Some of these algorithms are also applied to real humanoid robots so that they can learn how to walk [22] or even score penalty kick for the RoboCup Standard Platform League [23].

To summarize, a lot of RL research has been applied to Games and Robotics. **The goal of this thesis paper is to propose a brand new benchmark of a 3D underactuated robot complementing the OpenAI Gym library, with a set of RL solutions to control it.**

5 Setting of the Computer Environment

The computer model of the robot was made using the *MuJoCo* software. MuJoCo stands for "**Mu**lti-**Jo**int dynamics with **Co**ntact" and is a software simulating contacts between objects [24], and in the case of this thesis, the interaction between the robot and the floor. The simulation environment was written in *Python* and the link between MuJoCo and Python is done through the package *mujoco-py*.

However, getting MuJoCo set up can be tedious and time consuming because certain versions of MuJoCo are only compatible with certain versions of Python and mujoco-py. Reference [25] gives all the steps to follow in order to easily set up MuJoCo.

The following combination was used for this thesis:

- Ubuntu *16.04*
- Python *3.5.2*
- MuJoCo *mjpro150*
- mujoco-py *1.50.2*

6 Computer Model of the Agent

The model of the agent was written in a XML file, since this is the only type of file supported by MuJoCo. The XML code of the final version can be found in Appendix A. To make a model as close to reality as possible, the dimensions were measured directly on the Katita itself.

6.1 Dimensions

Figure 10 presents the dimensions and angles used in the XML model.

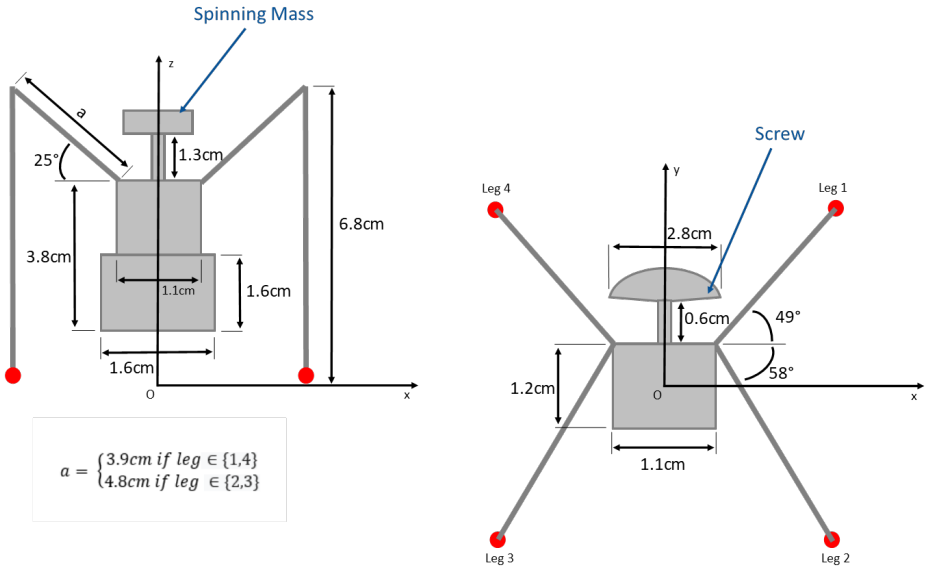


Figure 10: Dimensions used in the XML file to model the robot. The figure on the right shows a top view of the Katita without representing the spinning mass. The figure on the left shows a front view

In the XML file, each shape is represented by a geometry: two boxes for the core body, cylinders for the legs, spheres for the feet and ellipsoids for the spinning mass and the screw.

6.2 Materials and characteristics

One important parameter to set in the XML file A is the density of each geometry defined. This was done by looking at the materials composing the Katita.

The core of the body, the gears, the legs and the mass are made of stainless steel, and the red feet are in plastic [26]. These materials have the following characteristics:

- Stainless steel has a density of 7700kg/m^3 [27].
- The density of plastic is set to 1100kg/m^3 by taking the average of different types of plastic [28] [29].

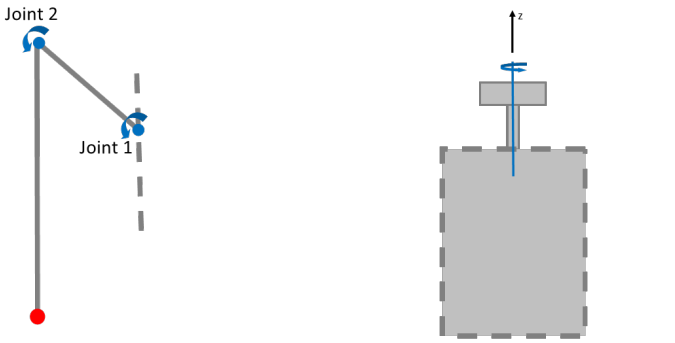
Moreover, the motion of the Katita is enabled by the elasticity of the stainless-steel legs. Pressing on the legs make them bend and act like springs, which make the whole robot jump. This elasticity was modelled by using *Joints*.

6.3 Joints

In MuJoCo, the motion between bodies can be allowed by setting joints. A joint defined in a body connects the parent to the child body, allowing Degrees Of Freedom (DOF) depending on its type. In the model, ten joints are defined:

- 1 *free* joint applied on the whole robot that enables 6 DOF: 3 translations and 3 rotations.
- 1 *hinge* joint for the rotation of the mass around the z axis - illustrated in figure 11(b).

- To model the elasticity of the legs, a good approximation is to define 2 *hinge* joints per leg, one at each corner, coupled with a spring that counters the rotation. The joints axes are illustrated in figure 11(a) by the blue dots, the consequent rotation by the blue arrows.



(a) Leg joints in blue circle (b) Mass joint in blue axis

Figure 11: Illustration of the joints to simulate the elasticity of the legs

The springs of the joints in the legs have two characteristics to be set in MuJoCo: a stiffness and a damping coefficient. **The definition of these coefficients was done in collaboration with Aaron Snoswell.** The motion of the actual robot was recorded with a high-speed camera in order to observe the oscillations of the legs. Figure 12 shows its behaviour for a complete rotation of the mass.

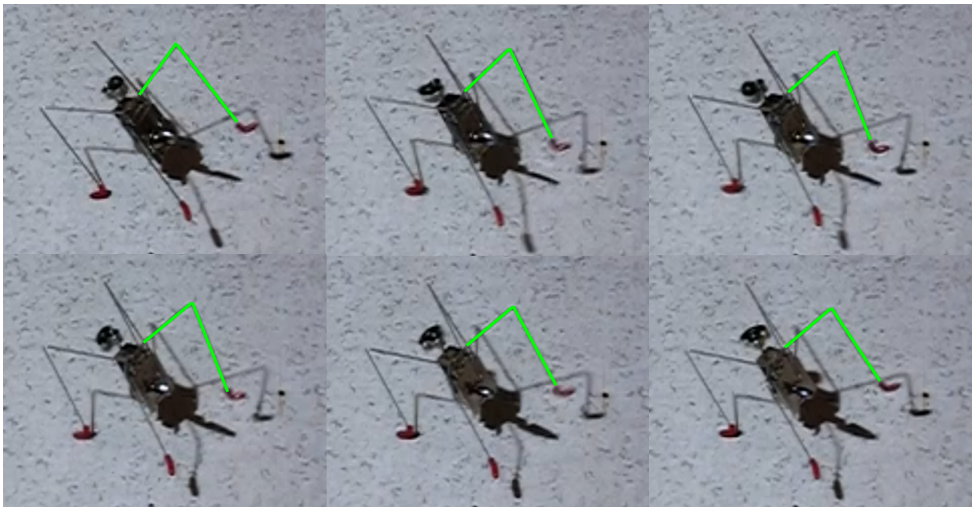


Figure 12: Frames extracted from a slow motion video showing the oscillations of the legs of the robot for a complete rotation of the mass. The camera used records 480 frames per second.

By looking at the behaviour of the robot in slow motion, it appears that the mass makes the legs oscillate while spinning. For example, the one highlighted in green in figure 12 is stretched on the first frame (top left corner), then compressed and stretched again. To replicate this behaviour on the computer model, several stiffnesses and damping coefficients were tried until the motion of the model was visually close enough to reality. **In the end, a stiffness of $K = 0.4$ N/m and a damping coefficient equal to $d = 1e^{-5}$ for each joint lead to the motion illustrated in figure 13.**

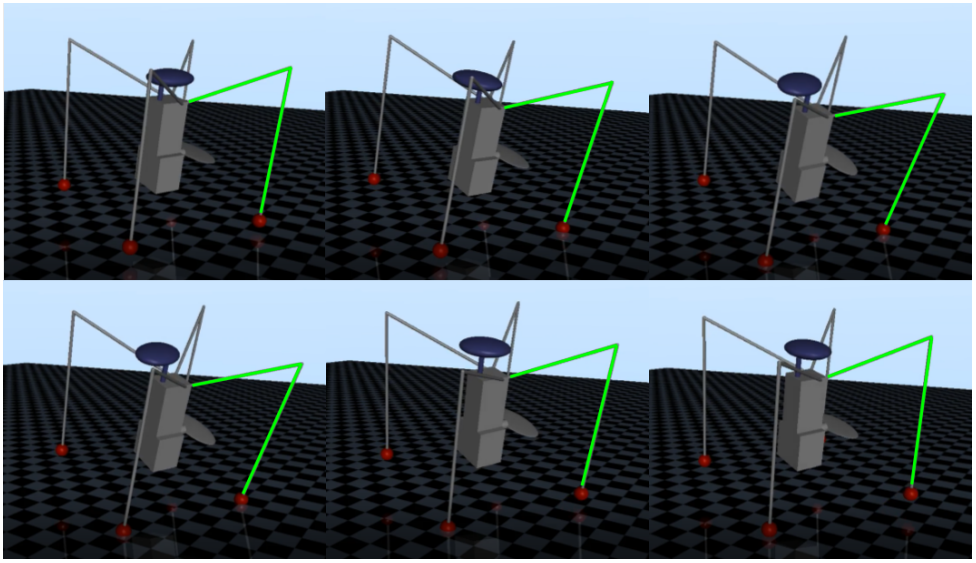


Figure 13: Frames extracted from MuJoCo showing the oscillations of the legs of the computer model for a complete rotation of the mass.

The computer model has a similar motion as the one shown on figure 12, with a leg that stretches, compresses and stretches again within one spin of the mass.

6.4 Definition of the variables

Figure 14 defines all the angles and coordinates.

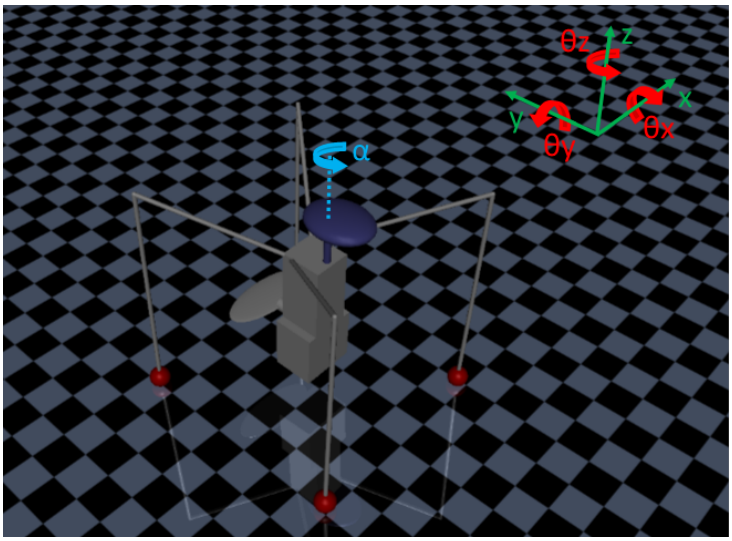


Figure 14: Display of the robot model via MuJoCo

The following notations are used in this report:

- x, y, z: cartesian coordinates of the robot.
- $\theta_X, \theta_Y, \theta_Z$: roll, pitch and yaw angles.
- α: the rotation angle of the mass.

7 Architecture of the code

The simulation code is split into three main classes: one for the agent, one for the environment and one to run the experiment. More precisely, figure 15 shows the architecture of the code through an UML diagram.

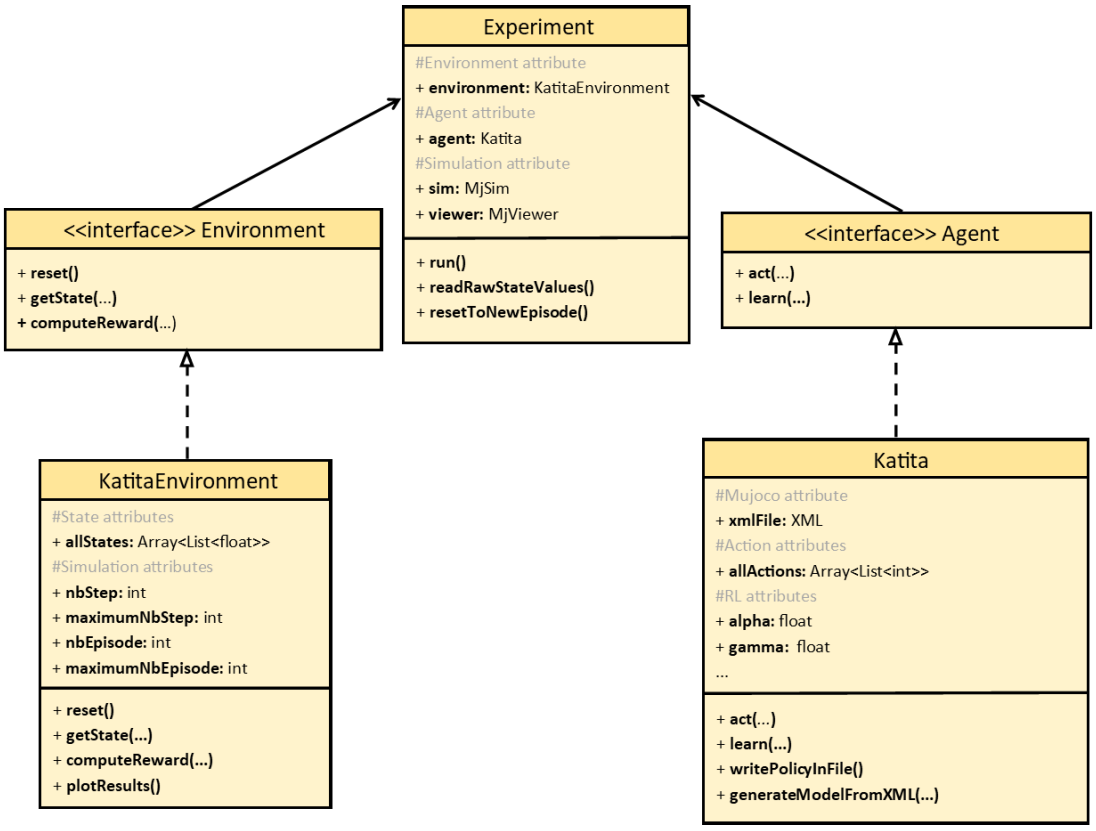


Figure 15: UML Diagram of the Simulation Code. For comprehension reasons, only the most relevant attributes and methods are shown. Each box represents a class or an interface, and each class has a set of attributes and methods.

KatitaEnvironment class
This is where the RL environment is modelled. It implements the interface *Environment* and overrides its methods:

- `reset()`: resets the environment parameters at the end of each episode.
- `getState()`: returns the state of the agent.
- `computeReward()`: computes the reward given to the agent for taking an action.

The simulation is divided into episodes, each of them is divided into steps. **One action corresponds to one step.** After a given number of steps - defined by the attribute `maximumNbStep` - the current episode finishes, the agent comes back to its initial position and the next one starts. The simulation finishes when `nbEpisode` reaches the maximum number of episodes set by the user through the attribute `maximumNbEpisode`.

Katita class
This is where the agent is modelled and where the RL algorithms are applied. It implements the interface *Agent* and overrides its methods:

- `act()`: takes an action based on the state the agent is in.
- `learn()`: updates the approximation of the policy/value function based on the past states, actions and rewards encountered, using the RL attributes `alpha` and `gamma`.

Moreover, the *Katita* class owns the XML file of the robot - attribute `xmlFile` - and generates a MuJoCo model through the method `generateModelFromXML()`. Once a simulation is done, the policy/value function approximation is saved in a text file through `writePolicyInFile()`.

Experiment class
This is where the simulation is run and where the instances of the *Katita* and *KatitaEnvironment* classes are generated, respectively named `agent` and `environment`. The main methods are:

- `run()`: runs the simulation.
- `readRawStateValues()`: returns the positions and velocities measured by MuJoCo at each step. These values are then sent to `environment.getState(...)` to be converted into states.
- `resetToNewEpisode()`: resets the simulation parameters at the end of each episode.

Such a code architecture is flexible enough to implement both the discrete and continuous approaches.

8 Discretization of the State and Action Spaces

The first approach to make the agent learn how to move straight is to simplify the problem representation by discretizing the action and state spaces. This enables the implementation of simple RL algorithms like *Q-Learning* (see 7) or *SARSA(λ)* (see 6). Then, the learning process can be simplified as well by slightly modifying the goal: **the direction the agent has to follow is set to the positive x axis** so that the problem becomes invariant in x.

Knowing this, the robot learning how to move straight becomes a symmetrical problem about the x axis. This symmetry can then be used to reduce the state space dimension.

8.1 State Representation

The position of the agent in time is defined by six parameters: three cartesian coordinates (x, y, z) and three angles $(\theta_X, \theta_Y, \theta_Z)$. However, discretizing all of them to define the state space would lead to a huge number of states (several millions!). Therefore, some of them are set to 0 by making the robot wait to be still before taking an action. In this case, $\theta_Y = \theta_Z = z = 0$. Moreover, as the problem is invariant in x, **states can be fully defined using only two parameters: the y coordinate and the yaw θ_Z.**

Thus, the state space can be discretized as follow:

- $y \in \{-3, -2.5, -2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2, 2.5, 3\}$ [in cm]. If the agent happens to go beyond the boundaries while training, i.e. $y < -3cm$ or $y > 3cm$, the simulation resets to a new episode and the robot starts from its initial position again.

- $\theta_Z \in \{0, 20, 40, ..., 340\}$ [in °].

There are 13 y values and 18 angles, so 234 states in total. However, the symmetry of the problem can be exploited to reduce even more the state space dimension.

8.2 Symmetry with respect to the x axis

Figure 16 illustrates the symmetry with respect to the x axis.

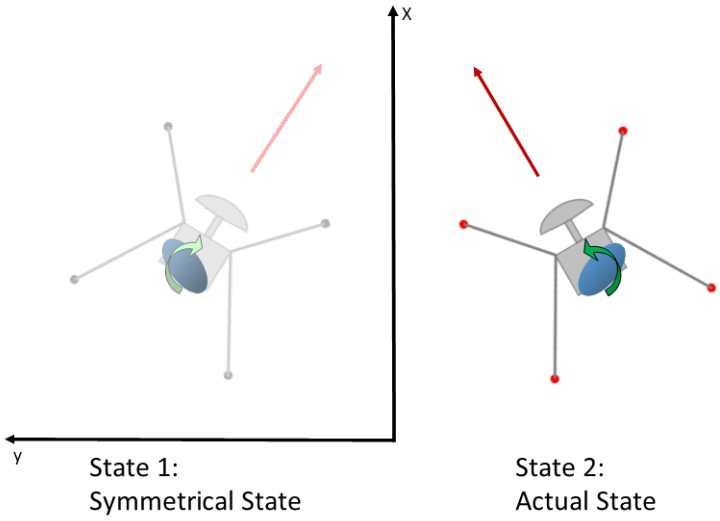


Figure 16: Illustration of the symmetry of the problem. On the right is the actual robot, and on the left is the symmetrical state with respect to the x axis.

When the agent is in state 2 (y_2, θ_2) with $y_2 < 0$, the optimal action A_2 to move towards the positive x axis is to spin the mass so that x increases and y decreases towards 0. On the other hand, if the robot is in state 1 (y_1, θ_1), the optimal action that leads to the same Δx and Δy is symmetrical to A_2. Knowing this property, the state space can be reduced in half by only considering when the agent is in the half-plane $(y > 0)$. When $y < 0$, the optimal action can be found by taking the symmetrical action of the symmetrical state.

Therefore, it is enough to consider $y \in \{0, 0.5, 1, 1.5, 2, 2.5, 3\}$, leading to only 126 states.

8.3 Action Representation

The motion of the robot is obtained by spinning the mass. In a discrete action space, three parameters influence the movement between 2 states:

- The angle α_{start} for which the mass starts spinning.
- The angle α_{end} for which the mass stops spinning.
- The velocity v.

To decrease the number of actions, α_{end} was omitted, and an action is only defined by a 360° rotation from α_{start}. Thus, the action space can be discretized as follow:

- $\alpha_{start} \in \{0, 10, 20, ..., 350\}$ [in °].

- $v \in \{-85, 85\}$ [in rad/s].

There are 36 angles and 2 velocities, **so 72 actions in total.**

Such a representation of states and actions is a Markov Decision Process. Indeed, the decision of the action to take can be made only by looking at the current state (y, θ_Z), without needing the full history of the agent behaviour.

8.4 Summary of the Process

The agent starts still in its initial position $(y, \theta_Z) = (0, 0)$. It chooses an action to take defined by a mass angle α_{start} and a velocity v. The mass moves slowly from the initial angle to the desired α_{start}. Note that at this point, the mass has to move slowly enough not to make the robot move. When the mass reaches α_{start}, the action is taken by making it spin at v for one complete turn. Then, the agent gets a reward and waits to be still again before taking another action.

8.5 Reward Function

The reward function used is:

$$reward = \frac{\Delta x}{\sigma\sqrt{2\pi}} \exp(-\frac{1}{2}(\frac{y-\mu}{\sigma})^2) \tag{23}$$

With $\begin{cases} \Delta x = x_{afterAction} - x_{beforeAction} : \text{the x distance between the 2 states} \\ y : y\text{-coordinate of the state after taking the action} \\ \sigma = 0.85 \\ \mu = 0 \end{cases}$

Figure 17 shows the corresponding $\frac{reward}{\Delta x}$ distribution against y.

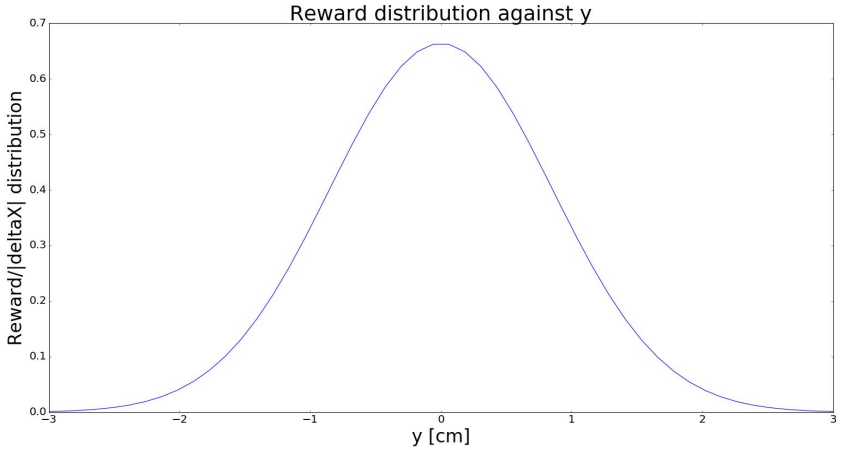


Figure 17: Reward distribution against y

The agent is rewarded positively when $\Delta x > 0$ and negatively when $\Delta x < 0$. The purpose of the normal distribution over y is to encourage the robot to stay close to the axis $y = 0$ while moving towards the positive x. Therefore, such a reward makes the agent understand how to act to move along the positive x axis.

8.6 Implementation of Q-Learning

Such low-dimensional discrete state and action spaces enable the implementation of the Q-Learning algorithm.

8.6.1 Default parameters

The training is done over 200 episodes of 50 steps, 1 step being one action. The exploration is done by using the ϵ-greedy method: at each step, the agent either exploits the Q-Learning table with probability $1 - \epsilon$ or explores random actions with probability ϵ. To make the robot converge to an optimal set of actions by the end of the training, the ϵ value decreases over the number of episodes. This means that at first, it explores a lot to assess the value of taking actions in given states, and at the end, it exploits a lot the optimal actions. The corresponding ϵ function is:

$$epsilon = min(1, \frac{4}{numberEpisode}) \tag{24}$$

In addition, as explained in section 8.3, the default action is to spin the mass by 360° from a starting angle α_{start} at a speed v. The results of such a training are shown in figure 18.

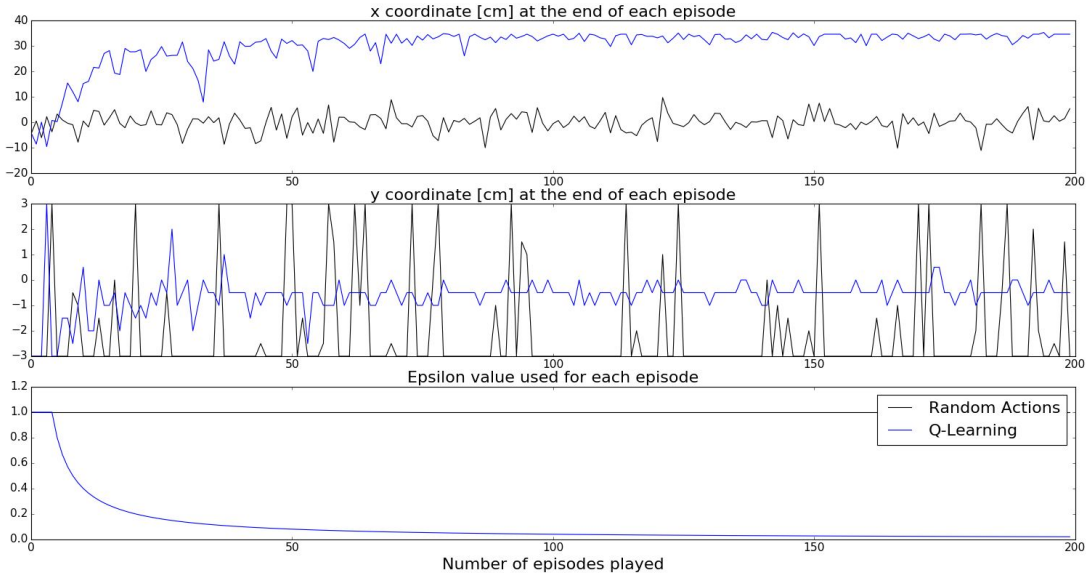


Figure 18: Learning of the agent using Q-Learning for 200 episodes of 50 steps. The x axis of the 3 plots is the number of episodes played. The first and second graphs draw the final x and y of the robot at the end of each episode. The third graph shows the variation of the ϵ parameter for exploration.

The black curves are the result of taking random actions all the time without learning and the blue ones come from the implementation of Q-Learning. Firstly, figure 18 shows that the agent often goes beyond the y boundaries by taking only random actions and does not manage to go towards the positive x. However, the blue x and y plots show that the agent starts to learn within the first 20 episodes since the x coordinate increases from 0 to 10cm. The agent converges to an optimal behaviour after the 70^{th} episode when it stops going beyond the y boundaries. The final distance reached within an episode of 50 steps is 19.36cm.

Nevertheless, one can wonder if the default action defined previously is optimal and enables the agent to go as far as possible. What if spinning the mass by a lower or higher angle makes it go further than 19.36cm?

8.6.2 Determination of the Optimal Rotation Angle

As mentioned in section 8.3, an action has been defined by default by a start angle α_{start}, a velocity and a 360° spin. However, 360° is arbitrary and might not be the angle

that leads to the best performances. **The optimal rotation angle is defined as the one that makes the agent move the fastest towards the positive x after learning a policy.** Therefore, it can be found as follow:

- Test 5 rotation angles: 180°, 360°, 540°, 720° and 900°.
- For each of these angles, train the agent over 200 episodes of 50 steps to find an approximation of the Q-value function.
- Exploit this Q-table over 1 episode and measure the speed v_x of the agent moving towards the positive x.
- Repeat the experiment 5 times for each angle and average the speeds.

The results are summarized in figure 19.

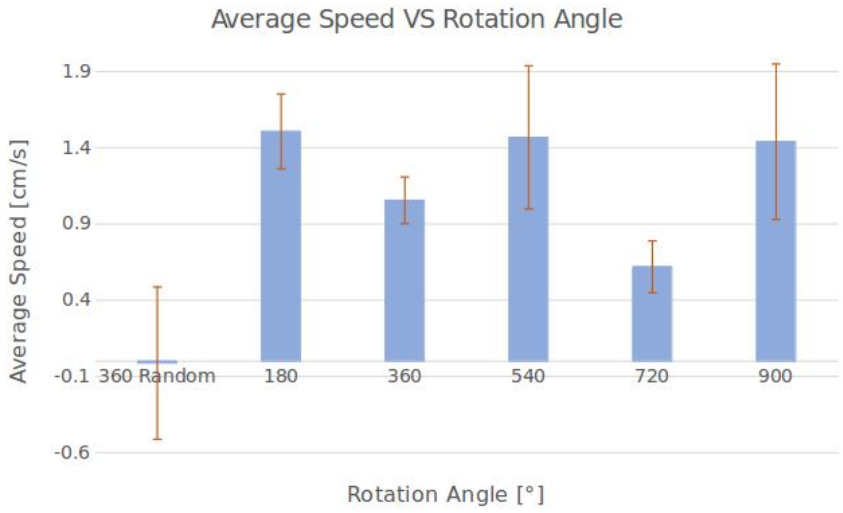


Figure 19: Plot of the average speed over 5 experiments for each rotation angle of the mass. The first bar is used as reference. It represents the average speed and uncertainty of the agent taking only random actions with a spinning angle equal to 360°.

Figure 19 shows the average speed of the robot and the uncertainties for each spinning angles. The details of uncertainty calculations can be found in Appendix B.

Firstly, the bar chart highlights the effect of Q-Learning on the speed and variance: while taking random actions leads to an average speed close to 0 with high variance, Q-Learning enables to get positive speeds and to lower uncertainties. Secondly 180°, 540° and 900° are the angles that lead to the highest average speed - about 1.5cm/s. However, 180° seems to make the learning more consistent since it is the one that shows the lowest variance. This can also be observed by looking at the learning curves in figure 20.

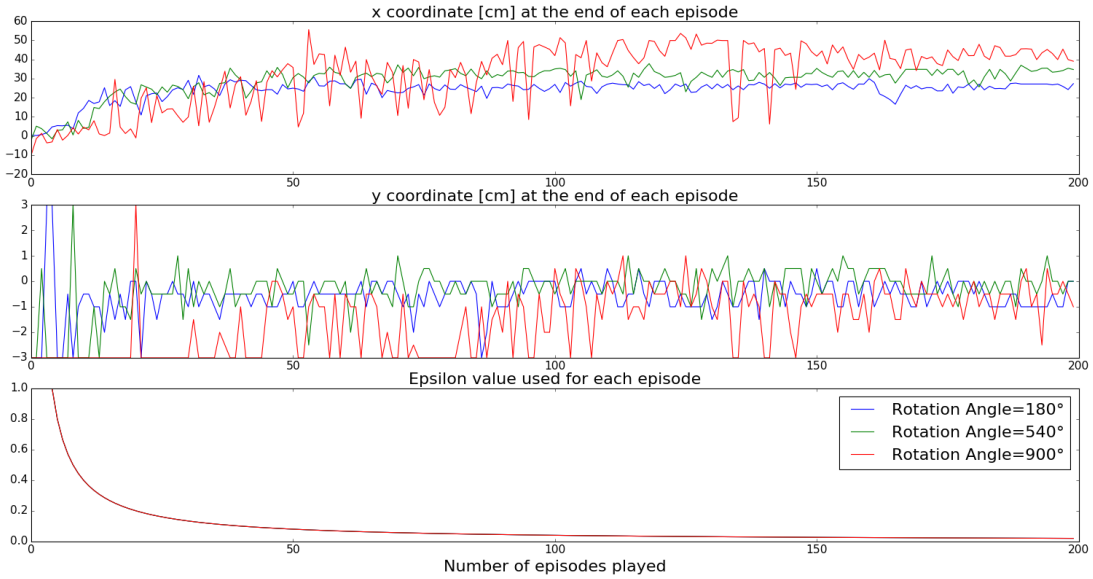


Figure 20: Comparison of the learning curves for a spinning angle of the mass equal to 180°, 540° and 900°

The high variance of the 900° case previously observed in figure 19 can be seen on the red learning curve as well: the agent struggles to converge to an optimal behaviour and keeps on going beyond the y boundaries even after the 140^{th} episode.

Therefore, the optimal rotation angle that leads to the highest v_x and the most consistent learning is 180°.

8.7 Implementation of SARSA(λ)

As explained in section 3, SARSA(λ) is an On-Policy Control algorithm aiming at approximating the optimal state-action value function q_π. It is very suitable for low-dimension state and action spaces.

8.7.1 Default Parameters

Similarly to Q-Learning, the training is done over 200 episodes of 50 steps and using the same ϵ function defined in equation 24 for exploration. The rotation angle for each action is set to 180° as a result of the study carried out in section 8.6.2. However, SARSA(λ) involves the use of another coefficient $\lambda \in [0, 1[$. The next section aims at finding the best λ that leads to an optimal learning.

8.7.2 Determination of the Optimal Lambda

Figure 21 shows the learning curves of the agent for three different λ values.

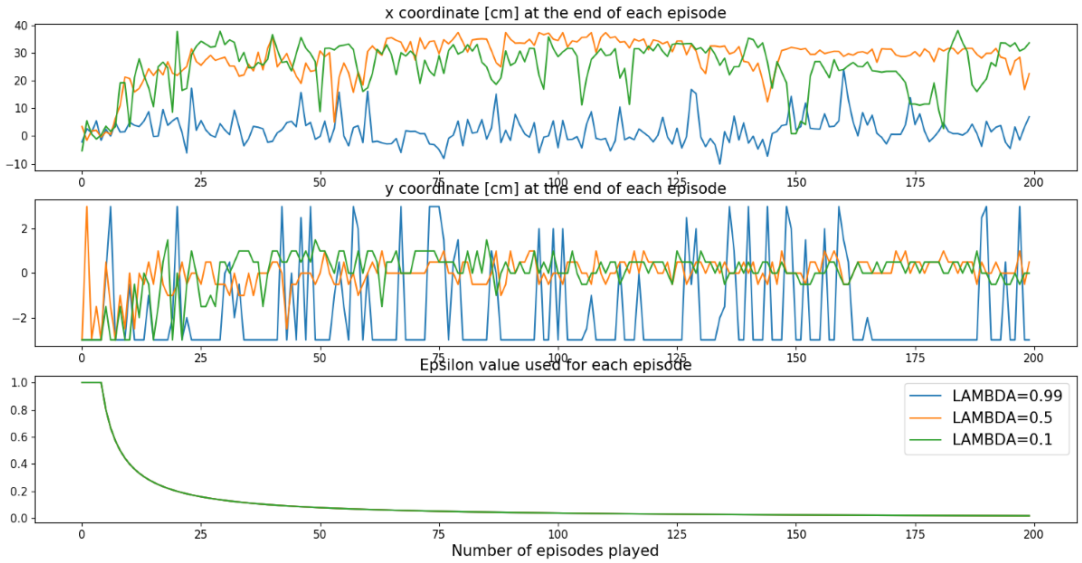


Figure 21: Comparison of the learning curves with λ equal to 0.99, 0.5 and 0.1

The blue curve shows that the agent does not manage to learn for $\lambda = 0.99$ and seems to take random actions since the average x distance reached after 200 episodes is close to 0. On the other hand, the robot seems to learn with $\lambda = 0.1$ and $\lambda = 0.5$ because the x distance increases over the number of episodes. However, it struggles to learn with $\lambda = 0.1$ because the x distance keeps on varying a lot for low exploration - i.e. low ϵ. Therefore, one can wonder what is the optimal λ value that enables the best learning?

The learning is optimal when it leads to the greatest x value with the lowest variance, i.e. when the learning curve goes as high and flat as possible over the number of episodes played. A way to quantify the quality of the learning is to compute the mean and variance of the x distance after the 50^{th} episode, which is approximately when the curve starts reaching a more steady state. Thus, the optimal λ can be determined as follow:

- Run one training simulation for each $\lambda \in \{0.001, 0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 0.99\}$ with the default parameters defined in section 8.7.1.
- For each simulation, compute the mean and the variance of the x distances reached after the 50^{th} episode.
- The optimal λ is the one that leads to the highest mean and lowest variance.

Figure 22 shows the results.

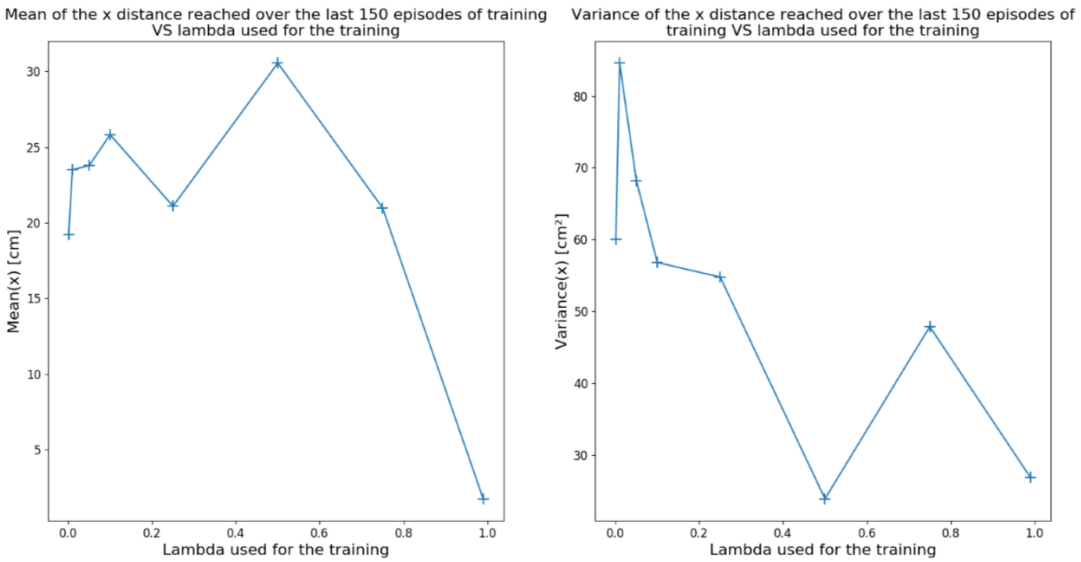


Figure 22: Determination of the optimal λ. The left graph plots the mean of the x distances reached during the last 150 episodes of training for a given λ. The right graph plots the variance.

As observed previously, the agent does not manage to learn with $\lambda = 0.99$ since the mean distance reached is only 1.76cm. **On the other hand, $\lambda = 0.5$ seems to be optimal because it leads to the highest mean distance (30.55cm) and the lowest variance (23.95cm²).**

Now that the optimal λ is known, it is interesting to compare the performances of SARSA(λ) with Q-Learning.

8.8 Q-Learning and SARSA(λ) Comparison

Figure 23 compares the learning curves of Q-Learning and SARSA(λ) with $\lambda = 0.5$. Both training were carried out with the same ϵ function and a spinning angle equal to 180°.

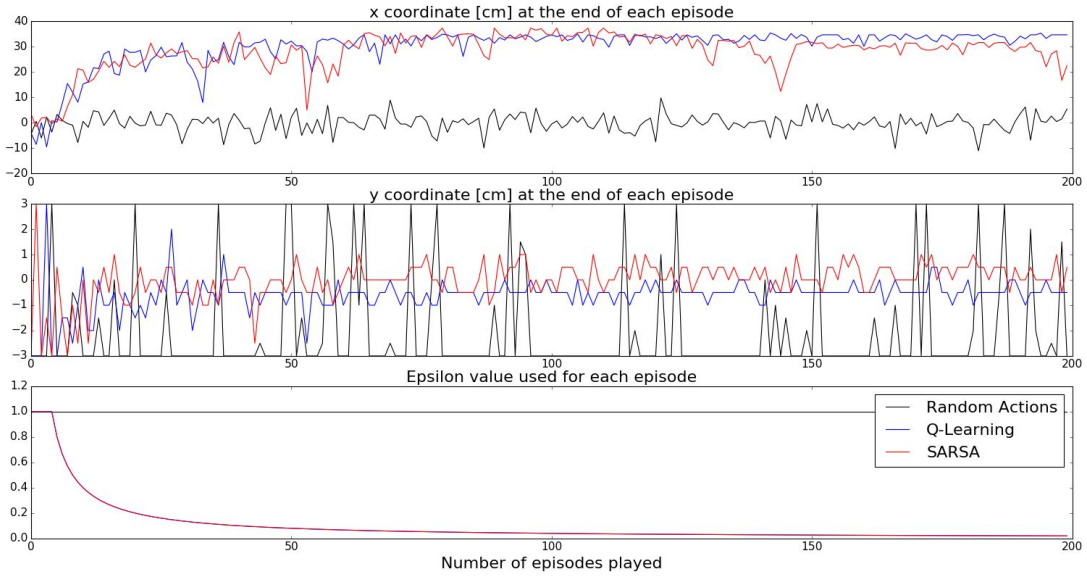


Figure 23: Learning curves of the Q-Learning and SARSA(λ) algorithms

The overlap of the blue and red x curves show that the two algorithms converge to the same optimal policy, with slightly more variance for SARSA(λ).

After training for 200 episodes, the policy found can then be exploited to assess how well the agent completes the desired task. Figure 24 plots the trajectories of the agent over 1 episode of 50 steps obtained by exploiting the trained policy.

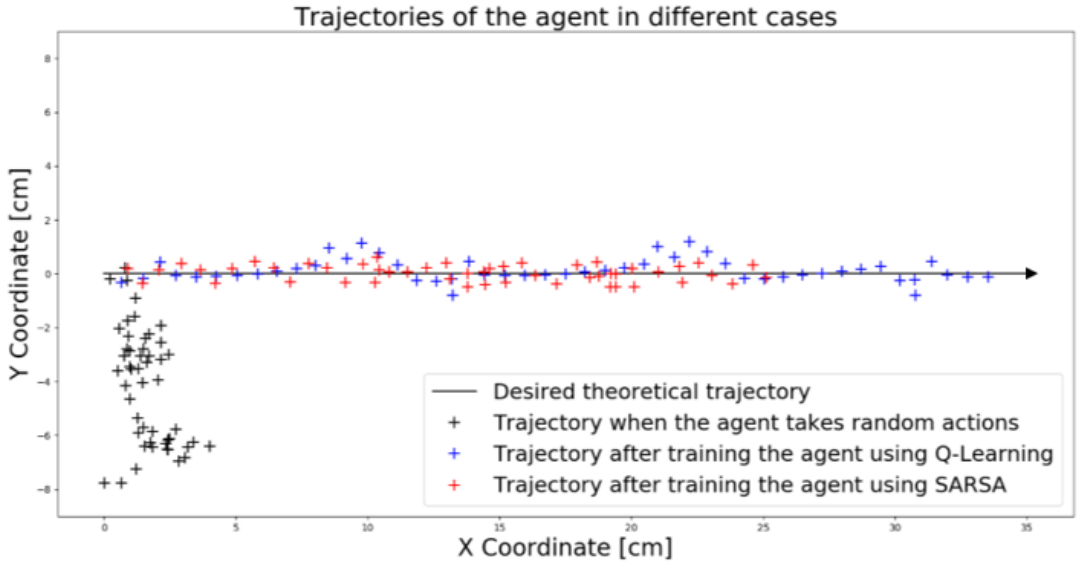


Figure 24: Trajectories of the agent using the policies found with Q-Learning and SARSA(λ) algorithms

Figure 24 shows a "view from the top" of the agent interacting with the environment. The dots represent the position of the robot at each time step of the episode. As a reference, the black trajectory is the result of taking only random actions.

One can see that the policies found with Q-Learning and SARSA(λ) enable the robot to move along the x axis quite accurately, with only slight variations towards the y axis. It seems that the agent goes further with Q-Learning than with SARSA(λ). The speed corresponding to the blue trajectory is $v_x = 1.37cm/s$.

However, this speed strongly depends on the representation of the problem. Discretizing the state space forces the agent to wait to be still before taking an action and decreases its overall speed. Therefore, one can wonder if considering a continuous state space could be more convenient and could make the robot move faster.

9 Continuous State Space

The second approach to control the movement of the robot is to treat the problem as a continuous problem since the state space that describes the position of the agent over time is continuous. In this case, it is not possible to work with a Q-table anymore because there are an infinite number of states. The solution is to learn directly the policy without considering the Q-value function and implement the REINFORCE algorithm (see section 3).

The main advantage of this method is that the agent does not have to wait to be still to take actions as it is moving in a continuous space. Indeed, the discretization presented in section 8 leads to a huge number of states, and the only way to reduce it is to simplify the problem by waiting for the agent to be still to take actions. However, REINFORCE

algorithm can handle continuous spaces with many state parameters, therefore θ_X and θ_Y can be considered to represent the position over time and the agent can take actions when tilted.

9.1 State Representation

As mentioned in section 8, the position of the robot is represented by six parameters: the cartesian coordinates (x, y, z) and the Euler angles $(\theta_X, \theta_Y, \theta_Z)$. The angle of the mass at each time step needs to be known as well through the parameter α. However, knowing the position is not enough in a continuous state space because the agent can take actions when moving. Thus, the direction in which it is moving has to be considered as well by calculating the derivative of each of the position parameters over time. The resulting state is the following vector:

$$s = (y, z, \theta_X, \theta_Y, \theta_Z, \alpha, \dot{x}, \dot{y}, \dot{z}, \dot{\theta_X}, \dot{\theta_Y}, \dot{\theta_Z}, \dot{\alpha}) \tag{25}$$

Note that the x coordinate is omitted because of the invariance in x (see section 8).

9.2 Action Representation

The action space is discrete and is represented by only one parameter: the velocity of the mass. One action corresponds to spinning the mass for 5° at a chosen speed. This means that every 5° of the rotation of the mass, the state is updated and a new velocity v is selected accordingly. The values v used are [in rad/s]:

$$v \in \{-85, -70, -50, -30, -10, 10, 30, 50, 70, 85\} \tag{26}$$

This representation allows the agent to spin the mass clockwise and anti-clockwise with different velocities in order to balance itself out and move towards the positive x axis. **It is a Markov Decision Process because the decision of taking an action only depends on the current state.**

9.3 Implementation of REINFORCE

The key to the REINFORCE algorithm is to find the right relationship to represent the policy, i.e. find the action/state mapping that leads to the selection of the optimal action for each state. This mapping is done through a set of parameters and the goal is to optimize them through algorithm 8 to converge to the optimal policy.

9.3.1 Action Selection

The policy is approximated by a Neural Network. A Neural Network is a set of nodes - called neurons - linked to each other via weighted connections [30]. An illustration of a Neural Network is given in Appendix C. The parameters of such a model are the weights and biases connecting each layer. Figure 25 shows how the actions are selected using this model.

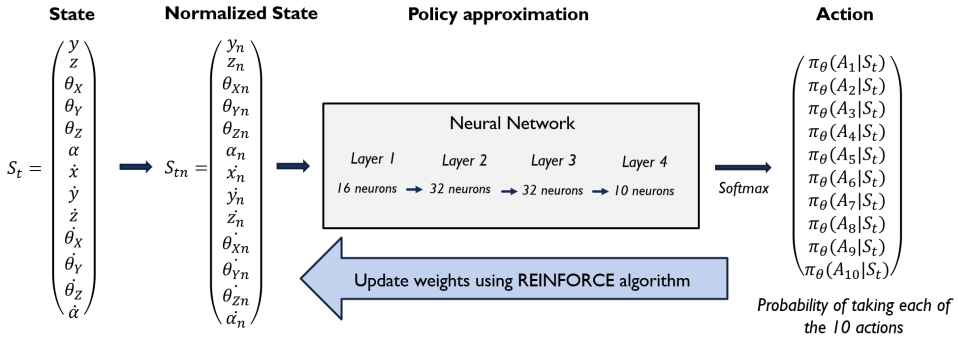


Figure 25: Principle of Action Selection using a Neural Network and REINFORCE algorithm

The process is the following:

- The agent is in a state S_t at time step t.
- The state S_t is normalized so that each of its parameters varies between $[-1, 1]$. This step is important to make sure that no parameter has more influence on the output action than others. The normalized state S_{tn} is the input of the Neural Network.
- From the state S_{tn}, several neurons get activated in the Neural Network. The output is a 10-neuron layer.
- The softmax function is then applied to the output layer in order to get probabilities - see Appendix D for more details. Therefore, the resulting array contains the probability of taking each of the 10 actions given the state S_t.
- The action is chosen according to these probabilities: the one with the highest probability has the highest chance of being selected.
- At the end of each episode, the weights and biases of the Neural Network are updated using the REINFORCE algorithm 8 in order to increase the probability of taking the actions that lead to high returns given a state S_t. At the end, the parameters of the Neural Network are expected to converge to the optimal policy where the probability of taking the optimal action given a state is equal to 1.

Note that the weights and biases are initialized randomly.

9.3.2 Results

The reward function used is the same as the one used in the discrete state space described in section 8.5. In terms of learning, this time the episodes have 4000 steps instead of 50. Indeed, the agent takes actions every 5° of rotation of the mass, and therefore needs more actions to be able to learn and travel the same distance as in discrete state space. Figure 26 shows the trajectory of the robot after learning in continuous state space using the REINFORCE algorithm.

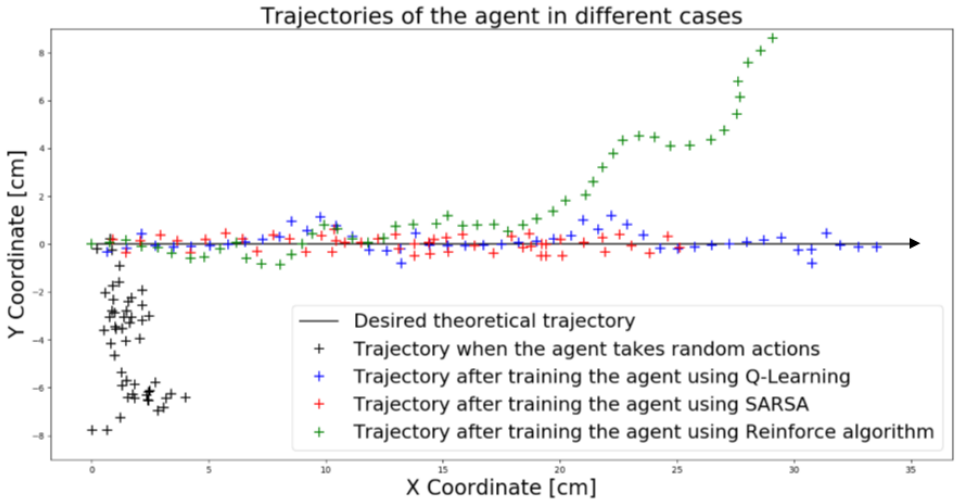


Figure 26: Trajectories of the agent using the policies found with Q-Learning, SARSA(λ) and REINFORCE algorithms

The green trajectory shows that the agent managed to learn how to move towards the positive x axis. It sticks to the axis quite accurately until $x = 18cm$ and then diverges towards the positive y. This is not surprising because the REINFORCE algorithm is known to have a high variance, which means that the agent struggles to converge to an optimal policy even after learning for many episodes [9]. This is why the final trajectory looks acceptable but not optimal compared to Q-Learning and SARSA(λ).

Moreover, the travelling speed measured for the green trajectory is $v = 1.56cm/s$. The agent moves faster than in discrete state space because it does not have to wait to take actions. **Therefore, it seems that there is a trade-off between speed and accuracy: the continuous state space enables the robot move slightly faster but less accurately. The learning algorithm needs to be improved to lower the variance.** This can be done using Baseline or Actor Critic methods [9].

10 Conclusion and Further Work

In this report, two approaches were presented to control the movement of the under-actuated robot: one discretizing the state space and the other treating the problem as a continuous problem.

The discretization enables the use of easy implementable algorithms such as Q-Learning and SARSA(λ). Such a representation aims at simplifying the problem in order to make the learning faster and easier. In the end, Q-Learning and SARSA(λ) show similar learning performances: the agent manages to learn and converge to an optimal policy in less than 200 episodes.

On the other hand, the continuous state space presents a more accurate representation of the problem, but more difficult to deal with. The high variance of the REINFORCE algorithm leads to a non-precise trajectory of the agent. It manages to move straight towards the positive x axis but then diverges away from the goal. However, a continuous representation enables the agent to move slightly faster since it does not have to wait to be in a discrete state to take actions.

Further work can be carried out to improve the learning and get even more interesting results. Firstly, the variance of the REINFORCE algorithm can be lowered by using Baseline and Actor Critic methods in order to obtain a straighter trajectory in the continuous

state space. Secondly, the representation of the problem can be even more accurate by using a continuous action space instead of discrete actions. This will enable the velocity of the mass to vary continuously in time, leading to a smoother motion of the agent. Thirdly, the ultimate goal is to make the physical version of the robot. The mechanical gears can be replaced by an electronic card to control the rotation of the mass using the algorithms developed on the computer model.

11 Acknowledgement

I would like to thank Dr Surya P.N. Singh for his precious support throughout the year. Reinforcement Learning was completely unfamiliar to me when I first started this thesis, and Dr Surya P.N. Singh helped me learn and apply the theory on this concrete project. I am very grateful for his patience and positivity.

I would also like to thank Mr Aaron Snoswell who helped me improve the computer model of the robot and helped me implement the algorithm in continuous state space.

References

[1] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. *Artificial intelligence*, 134(1-2), 57-83.

[2] Reuters News Agency. (2017, May 24). Computer beats Chinese master in ancient board game of Go. *The Telegraph*. Retrieved from https://www.telegraph.co.uk/news/2017/05/24/computer-beats-chinese-master-ancient-board-game-go/.

[3] Sutton, R. S., & Barto, A. G. (2017). *Reinforcement learning: An introduction*. MIT press.

[4] Yuan, Q., & Xiao, N. (2018). A monotonic policy optimization algorithm for high-dimensional continuous control problem in 3D MuJoCo. *Multimedia Tools and Applications*, 1-16.

[5] Swaminathan, A. (n.d). *Bandits* [PowerPoint slides]. Retrieved from Microsoft:DAT257x, edX, https://courses.edx.org/courses/course-v1:Microsoft+DAT257x+2T2018/course/.

[6] Fernandez, R. (n.d). *The Reinforcement Learning Problem* [PowerPoint slides]. Retrieved from Microsoft:DAT257x, edX, https://courses.edx.org/courses/course-v1:Microsoft+DAT257x+2T2018/course/.

[7] Fernandez, R. (n.d). *Dynamic Programming* [PowerPoint slides]. Retrieved from Microsoft:DAT257x, edX, https://courses.edx.org/courses/course-v1:Microsoft+DAT257x+2T2018/course/.

[8] Tran, K. (n.d). *Temporal Difference Learning* [PowerPoint slides]. Retrieved from Microsoft:DAT257x, edX, https://courses.edx.org/courses/course-v1:Microsoft+DAT257x+2T2018/course/.

[9] Hausknecht, M. (n.d). *Policy Gradient and Actor Critic* [PowerPoint slides]. Retrieved from Microsoft:DAT257x, edX, https://courses.edx.org/courses/course-v1:Microsoft+DAT257x+2T2018/course/.

[10] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

[11] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of go without human knowledge. *Nature, 550*(7676), 354.

[12] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[13] Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., & Tsang, J. (2017). Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems* (pp. 5392-5402).

[14] Nagendra, S., Podila, N., Ugarakhod, R., & George, K. (2017, September). Comparison of reinforcement learning algorithms applied to the cart-pole problem. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)* (pp. 26-32). IEEE.

[15] Zimmer, M., & Doncieux, S. (2018). Bootstrapping q-learning for robotics from neuro-evolution results. *IEEE Transactions on Cognitive and Developmental Systems, 10*(1), 102-119.

[16] Zamora, I., Lopez, N. G., Vilches, V. M., & Cordero, A. H. (2016). Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo. *arXiv preprint arXiv:1608.05742.*

[17] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971.*

[18] Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., ... & Silver, D. (2017). Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286.*

[19] Garg, U. (2018). Virtual Robot Climbing using Reinforcement Learning.

[20] Vuong, Q., Zhang, Y., & Ross, K. W. (2018). Supervised Policy Update for Deep Reinforcement Learning.

[21] Iglesias, R., Regueiro, C. V., Correa, J., & Barro, S. (1998, June). Supervised reinforcement learning: Application to a wall following behaviour in a mobile robot. *In International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems* (pp. 300-309). Springer, Berlin, Heidelberg.

[22] Peters, J., Vijayakumar, S., & Schaal, S. (2003, September). Reinforcement learning for humanoid robotics. In *Proceedings of the third IEEE-RAS international conference on humanoid robots* (pp. 1-20).

[23] Hester, T., Quinlan, M., & Stone, P. (2010, May). Generalized model learning for reinforcement learning on a humanoid robot. In *2010 IEEE International Conference on Robotics and Automation* (pp. 2369-2374). IEEE.

[24] MuJoCo. Advanced physics simulation. (n.d.). Retrieved from `http://www.mujoco.org/`.

[25] Balakrishnan, G.P. (2018). Setting up Mujoco [LinkedIn page]. Retrieved September 2, 2018 from `https://www.linkedin.com/pulse/setting-up-mujoco-ganesh-prasanna`.

[26] Kikkerland Katita Windup, Assorted Colors. *e-Commerce: online shopping.* [online] amazon.com. Available at: `https://www.amazon.com/Kikkerland-Katita-Windup-Assorted-Colors/dp/B001BJE532`. [Accessed 16 August 2018].

[27] Metals and Alloys - Densities. (n.d.). Retrieved from `https://www.engineeringtoolbox.com/metal-alloys-densities-d_50.html`.

[28] Plastic Properties Table. (n.d.). Retrieved from `http://www.dotmar.com.au/propertiestables.php`.

[29] Density of Plastic: Technical Properties. (n.d.). Retrieved from `https://omnexus.specialchem.com/polymer-properties/properties/density#values`.

[30] Marsland, S. (2011). *Machine learning: an algorithmic perspective*. Chapman and Hall/CRC.

[31] Mee, D. (2018). *The effect of multi-parametric experimental errors on computed engineering parameters* [Powerpoint slides]. Retrieved from ENGG7601, University of Queensland Blackboard Online, http://www.elearning.uq.edu.au/.

[32] Statistics How To. (2019). T-Distribution Table (One Tail and Two-Tails). Retrieved from https://www.statisticshowto.datasciencecentral.com/tables/t-distribution-table/.

Appendices

A Agent XML file

```
<mujoco model="KatitaV4">
    <!-- Units are SI -->

    <compiler coordinate="global"/>

    <option timestep="0.0002" />

    <default>
        <!-- Default density is steel at 7700kg/m^3 -->
        <geom rgba="1 1 1 1" density="7700" />
    </default>

    <asset>
        <!-- Get a nice blue sky -->
        <texture type="skybox" builtin="gradient" rgb1="1 1 1" rgb2="
           .6 .8 1" width="256" height="256"/>

        <!-- 1cm square checkerboard -->
        <texture name="texplane" type="2d" builtin="checker" rgb1="
           0.7 0.8 1" rgb2="0 0 0" width="512" height="512"/>
        <material name="checkerboard" reflectance=".3" texture="
           texplane" texrepeat="100 100" texuniform="true"/>
    </asset>

    <worldbody>
        <!-- ============= Light ============ -->
        <light pos="0 1 100" dir="0 1 1" diffuse="1 1 1"/>

        <!-- ============= 1m square floor with 1cm checks
           ============ -->
        <geom name="floor" pos="0 0 0" size="0.5 0.5 0.01" type="
           plane" material="checkerboard" />

        <!-- ============ Katita ============ -->
        <body name="katita" pos="0 0 0.013" quat="1 0 0 0">
            <joint name="wholeBody" type="free"/>

            <!-- ======= Core body ======= -->
            <!-- NB the boxes are not solid, hence we use a 10% and
               50% density estimate here -->
            <geom type="box" name="coreBody1" size=".0055 .006 .019"
               pos="0 0 .032" density="770" />
            <geom type="box" name="coreBody2" size=".008 .006 .008"
               pos="0 0 .021" density="3850" />
            <geom type="cylinder" name="screw1" size=".002" fromto="0
                0 .026  0 .012 .026" />
            <geom type="ellipsoid" name="screw2" pos="0 .017 .026"
               size=".014 .006 .0005" />
```

```
39
40              <!-- Front legs -->
41              <!-- ========= Leg 2 ========= -->
42              <body name="leg2upper">
43                  <geom type="cylinder" fromto=".0287 -.0308 .07 .005
                        .0035 .05" size=".00061" />
44                  <geom type="sphere" pos=".0287 -.0308 .07" size="
                        .00061" />
45                  <joint type="hinge" name="joint1Leg2" pos=".003 .0035
                         .049" axis="-2.32 -3.68 0" stiffness="0.4"
                        damping="0.00001" />
46
47                  <body name="leg2lower">
48                      <geom type="cylinder" fromto=".0287 -.0308 0.005
                            .0287 -.0308 .07" size=".00061" />
49                      <geom type="sphere" name="foot2" pos=".0287
                            -.0308 .005" size=".003" density="1100" rgba="
                            1 0 0 1" />
50                      <joint type="hinge" name="joint2Leg2" pos=".0287
                            -.0308 .068" axis="-2.32 -3.68 0" stiffness="
                            0.4" damping="0.00001" />
51                  </body>
52              </body>
53
54              <!-- ========= Leg 3 ========= -->
55              <body name="leg3upper">
56                  <geom type="cylinder" fromto="-.0287 -.0308 .07 -.005
                         .0035 .05" size=".00061" />
57                  <geom type="sphere" pos="-.0287 -.0308 .07" size="
                        .00061" />
58                  <joint type="hinge" name="joint1Leg3" pos="-.003
                        .0035 .049" axis="2.32 -3.68 0" stiffness="0.4"
                        damping="0.00001" />
59
60                  <body name="leg3lower">
61                      <geom type="cylinder" fromto="-.0287 -.0308 0.005
                             -.0287 -.0308 .07" size=".00061" />
62                      <geom type="sphere" name="foot3" pos="-.0287
                            -.0308 .005" size=".003" density="1100" rgba="
                            1 0 0 1" />
63                      <joint type="hinge" name="joint2Leg3" pos="-.0287
                             -.0308 .068" axis="2.32 -3.68 0" stiffness="
                            0.4" damping="0.00001" />
64                  </body>
65              </body>
66
67              <!-- Back legs -->
68              <!-- ========= Leg 1 ========= -->
69              <body name="leg1upper">
70                  <geom type="cylinder" fromto=".0287 .0328 .068 .003
                        .0035 .049" size=".00061" />
71                  <geom type="sphere" pos="0.0287 0.0328 0.068" size="
                        .00061" />
```

```
72                  <joint type="hinge" name="joint1Leg1" pos=".003 .0035
                        .049" axis="2.32 -2.68 0" stiffness="0.4" damping
                       ="0.00001" />
73
74                  <body name="leg1lower">
75                      <geom type="cylinder" fromto=".0287 .0328 0.003
                           .0287 .0328 .068" size=".00061" />
76                      <geom type="sphere" name="foot1" pos=".0287 .0328
                            .003" size=".003" density="1100" rgba="1 0 0
                           1" />
77                      <joint type="hinge" name="joint2Leg1" pos=".0287
                           .0328 .068" axis="2.32 -2.68 0" stiffness="0.4
                           " damping="0.00001" />
78                  </body>
79              </body>
80
81              <!-- ========= Leg 4 ========= -->
82              <body name="leg4upper">
83                  <geom type="cylinder" fromto="-.0287 .0328 .068 -.003
                        .0035 .049" size=".00061" />
84                  <geom type="sphere" pos="-.0287 .0328 .068" size="
                       .00061" />
85                  <joint type="hinge" name="joint1Leg4" pos="-.003
                       .0035 .049" axis="-2.32 -2.68 0" stiffness="0.4"
                       damping="0.00001" />
86
87                  <body name="leg4lower">
88                      <geom type="cylinder" fromto="-.0287 .0328 0.003
                           -.0287 .0328 .068" size=".00061" />
89                      <geom type="sphere" name="foot4" pos="-.0287
                           .0328 .003" size=".003" density="1100" rgba="1
                            0 0 1" />
90                      <joint type="hinge" name="joint2Leg4" pos="-.0287
                            .0328 .068" axis="-2.32 -2.68 0" stiffness="
                           0.4" damping="0.00001" />
91                  </body>
92              </body>
93
94              <!-- ========= Mass ========= -->
95              <body name="mass" pos="0 -.004 .061" quat="1 0 0 0" >
96                  <geom type="cylinder" name="threadMass" fromto="0
                       -.004 .05 0 -.004 .0625" size=".001" rgba=".4 .4
                       .8 1" />
97                  <geom type="ellipsoid" name="mass" pos=".0043 -.004
                       .061" size="0.008 0.01 0.003" rgba=".4 .4 .8 1" />
98                  <joint name="jointMass" type="hinge" pos="0 -.004 .05
                       " axis="0 0 1" />
99              </body>
100         </body>
101     </worldbody>
102 </mujoco>
```

B Measurements and Uncertainties of the Agent Speed

For a multi-sample experiment, the absolute uncertainty δv of the speed v can be calculated using the Student's law through equation 27 (considering no bias) [31]:

$$\delta v^2 = (\frac{t\sigma_v}{\sqrt{N}})^2 \tag{27}$$

With σ_v being the standard deviation of the N measurements of v, and t a coefficient coming from the Student's t-distribution. In this case, the experiment has $N = 5$ samples. For 95% confidence and $N - 1 = 4$ degrees of freedom, $t = 2.776$ [32].

The uncertainties are summarized in the following table 1.

Rotation Angle [°]	Simulation Number	x [cm]	T [s]	Speed v [cm/s]	Mean Speed v [cm/s]	δv [cm/s]
180	1	34.4	20.4	1.69	1.51	0.25
	2	33.72	21.9	1.54		
	3	35.43	28.6	1.24		
	4	32.39	23.5	1.38		
	5	36.42	21.5	1.69		
360	6	22.59	19.9	1.14	1.06	0.15
	7	23.89	22.3	1.07		
	8	19.32	21.2	0.91		
	9	26.3	21.8	1.21		
	10	24.94	26.2	0.95		
540	11	42.49	23.1	1.84	1.47	0.47
	12	25.45	25	1.02		
	13	41.38	23.1	1.79		
	14	42.52	27.1	1.57		
	15	30.5	27.1	1.13		
720	16	15.44	27	0.57	0.62	0.17
	17	18.58	27.8	0.67		
	18	13.64	30.1	0.45		
	19	15.53	26.8	0.58		
	20	23.03	28	0.82		
900	21	46.33	31.1	1.49	1.44	0.51
	22	46.7	29	1.61		
	23	42.03	30.4	1.38		
	24	23.17	29	0.80		
	25	54.88	28.6	1.92		

Table 1: Measurements of the agent speed for several rotation angles of the mass

C Neural Network

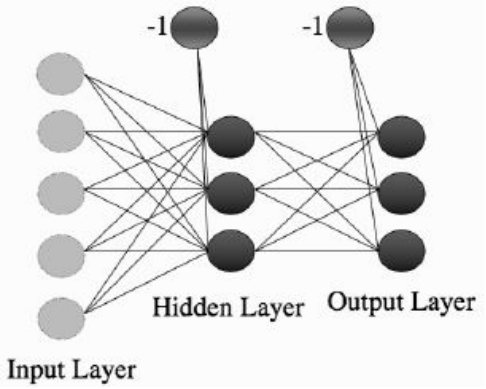


Figure 27: Illustration of a Neural Network with 5 inputs, 3 outputs and 1 hidden layer. The bias for every neuron is equal to -1 - [30]

A Neural Network is composed of inputs, outputs and hidden layers. Each neuron of a layer is linked to the ones of the adjacent layers via weighted connections [30].

Let $(x_k)_{k\in[\![1;m]\!]}$ be the inputs, $(h_k)_{k\in[\![1;n]\!]}$ the neurons of the hidden layer, $(w_{ij})_{i\in[\![1;n]\!],j\in[\![1;m]\!]}$ the weights connecting the inputs to the hidden layer and $(b_k)_{k\in[\![1;n]\!]}$ the biases. The way the value of a neuron is calculated is defined in equation 28 [30].

$$\forall k \in [\![1;n]\!], h_k = a(b_k + \sum_{j=1}^{m} w_{kj}x_j) \tag{28}$$

a is an activation function that adds a condition to the neuron to be activated [30]. **The activated function used in this thesis is the ReLu function defined as follow:**

$$\forall x \in \mathbb{R}, a(x) = max(0, x) \tag{29}$$

Neural Networks are usually used in Machine Learning to approximate nonlinear functions [3].

D Softmax function

The softmax function is widely used in Reinforcement Learning when the action space is discrete [3]. For any vector $z = (z_j)_{j \in [\![1;k]\!]}$, $k \in \mathbb{N}$, it is defined as follow [3]:

$$f(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}} \tag{30}$$

This function is really convenient because it can be interpreted as a probability:

- $\forall j \in [\![1;k]\!], 0 \leq f(z)_j \leq 1$
- $\sum_{j=1}^{k} f(z)_j = 1$

The main advantage of using the softmax function in Reinforcement Learning to select actions is that it can converge to a deterministic behaviour where the probability of taking the optimal action is equal to 1, and the other probabilities are nil [3].